A Programming Model for GPU-based Parallel Computing with Scalability and Abstraction

Balázs Domonkos
Budapest University of Technology and Economics, Hungary
Mediso Medical Imaging Systems, Hungary

Gábor Jakab
Mediso Medical Imaging Systems, Hungary

Abstract

In this paper, we present a multi-level programming model for recent GPU-based high performance computing systems. Involving cooperative stream threads and symmetric multiprocessing threads our model gives a computational framework that scales through multi-GPU environments to GPU-cluster systems. Instead of hiding the execution environment from the programmer using compiler extensions or metaprogramming techniques we aim a solution that both enables optimizations and provides abstract problem space mapping with code reusability and virtualization of hardware resources in order to decrease the programming effort. We evaluate an implementation of our model based on CUDA, OpenMP, and MPI2 technologies on a complex practical application scenario and discuss its performance scaling behavior.


Keywords: GPU-based computing, multi-GPU systems, GPU-based compute clusters

1 Introduction and Motivation

Although semiconductor capability increases at about the same rate for both CPUs and GPUs driven by advances in fabrication technology, computational performance of GPUs increases more rapidly than that of CPUs. With many transistors dedicated to instruction-level parallelism using branch prediction and out-of-order execution, CPUs are optimized for high performance sequential code execution. In contrast, graphics computations with rapid data-parallel characteristics result in higher arithmetic intensity for GPUs with the same transistor count. GPUs have undergone several stages of evolution in the last decade including increasing raw speed, precision, and programmability. From a highly graphics-related fixed point fixed-function pipelines, GPUs rapidly evolved through the era of dedicated vertex and fragment programs with limited SIMD programming features to general purpose fully programmable stream processors with data dependent branching, IEEE double precision data type support, and fulfilling a real Single Program Multiple Data (SPMD) programming model.

Meanwhile, various programming languages and frameworks have emerged supporting high performance data-parallel computations with real scatter operations and thread cooperation getting rid of former limited shared memory emulation in the texture memory. These continuous innovations, the wide deployment of GPUs, and the attractive performance per price ratio resulted increasing experimental research on GPUs which have become an attractive throughput architecture for high performance computing computations in various domains of science.

From the algorithmic point of view, computational tasks often contain exploitable concurrency that enables different parts of the problem to be solved simultaneously by multiple closely coupled processors (parallel computing). Usually, high-speed communication channels exist between these processing elements and fast synchronization is also possible. This allows compute-intensive problems to be solved in less time and also might enable large data-intensive problems to be handled. However, when memory resources are insufficient to store the model of a problem on a single device, the parallel program has to be further split up into loosely coupled instances that require only a limited amount of interaction during their execution (distributed computing). Commonly, there is no communication medium between these devices as fast as there is within a device and there is no adequate solution for fast synchronization.

It has been shown by many researches that GPUs and clusters of GPUs can be successfully applied for these purposes; see NVIDIA web site for various examples. However, when writing complex applications or software library stacks, one often faces issues of reusability, abstraction, or portability. On the other hand, the underlying computational efficiency can be lost by defining too high level, general, or excessively transparent GPU-frameworks. Therefore, we have investigated how an OpenCL-like model can be extended in two dimensions without losing the computational horse power: in the levels of scalability from a streaming device through an SMP computer to a cluster; and in the layers of abstraction from the hardware level through mapping the execution to the hardware to projecting the problem space to the execution (Figure 1).

The remainder of this paper is organized as follows. First existing languages, programming libraries, and frameworks are briefly summarized in Section 2. Considering the conjectural trends of the evolution, we introduce a simple formalism using which a multi-level and multi-layer programming model can be defined. Since code portability does not immediately involve efficient code reusability on different system architectures, our model does not hide the abstract hardware environment (Section 3) from the programmer using compiler extensions or metaprogramming techniques. Instead, we propose a solution that still enables architectural optimizations (Section 4) while also provides high-level mapping of the abstract problem space (Section 5). This supports code reusability and virtualization of hardware resources in order to decrease the programming effort. Although our primary contribution is our three-tier model, we present a potential software infrastructure and implementation based on CUDA, OpenMP, and MPI2 technologies (Section 6). Finally, to demonstrate the usability of the model, this implementation is evaluated on a complex practical application scenario, a tomographic reconstruction algorithm (Section 7) and its...
performance scaling behavior is discussed.

2 Previous Work

The concept to use the graphics hardware for general purpose algorithms has been introduced in the early nineties by Lengyel et al. by using a rasterization device for robot motion planning [Lengyel et al. 1990] and by Cabral et al. who accelerated tomographic reconstruction using a texture mapping hardware [Cabral et al. 1994]. Although, in that time most of the pioneering works have been done as proof of concept prototypes, from the beginning of this decade a clear new domain has been emerging for stream computations on the graphics hardware. Influenced by the GPGPU movement, numerous programming languages, libraries, and frameworks have been designed to make stream programming on the GPU easier and more productive. The common requirements for such frameworks were portability to the primary hardware and software platforms, and efficiency introducing only a low overhead compared to optimized hand-written implementations. On the other hand, high abstraction, that obscures the details of the underlying hardware was also welcome. Besides, a strong demand raised to couple the shader language and the host code into a single language to handle CPU-GPU interactions by grammar instead of using cumbersome API calls.

The evolution of stream programming on the GPU began with programmable shading using GPUs for their primary purpose: for doing computer graphics. The first high level real-time procedural shading languages were the Stanford Shading Language [Proudfoot et al. 2001], Cg [Mark et al. 2003], and graphics API specific languages like HLSL [Oneppo 2007] and GLSL [Rost et al. 2004].

In addition to the traditional rendering, GPGPU languages have appeared in order to support common activities like glue code generation and shader synthesis. Such languages are the RapidMind Development Platform [Montayne 2008] as the generalization of Sh [McCool et al. 2002; McCool et al. 2004], and the GPU++ language [Jansen 2007], that apply C++ meta programming techniques for generating syntax tree and synthesizing shader source code on-the-fly. Other approaches that use special compilers to generate kernel and host source code in compile time are the Brook stream programming language [Buck et al. 2004] and its spin-offs, PeakStream and Brook+. Other similar compiled languages are CGIs [Fritz et al. 2004], Accelerator [Tarditi et al. 2006], and Brahms [Bra 2006]. In addition there are GPGPU programming libraries like Shallows [Sha 2005] and Glift [Leifohn et al. 2006] which provide variety of standard data structures for the GPU shader model.

Closing the GPGPU era, currently general purpose GPU programming languages represent the most effective way to write and execute programs on GPUs without the need to map the algorithms onto a 3D graphics API such as OpenGL or DirectX. These solutions provide compilers, runtime libraries, and tools for efficient development and debugging. Frameworks of the main vendors are CUDA from NVIDIA [Nickolls et al. 2008], the CAL programming framework of AMD/ATI Stream [AMD/ATI 2008] (formerly Close To Metal), and Ct with Larrabee from Intel [Gholoum et al. 2008]. Designed especially for CUDA, CU-DASA [Strøgert et al. 2008] can handle multiple hardware levels in a CUDA-fashioned way. Also, there are languages that can compile source code for multiple hardware architectures, such as DirectX Compute Shaders, hmp [Hmp 2008] from CAPS Enterprise, OpenCL [Kronos OpenCL Working Group 2008], a new open standard for general purpose parallel programming from Kronos Group, and the C$ language [A. V. 2007]. Many of the models are restricted to only a single device. The domain of OpenCL and CUDA is a single host computer with possibly multiple devices. CUDA has similar purpose that our work has. However, the clear difference is that our model does not force the block-grid mechanism of CUDA to higher hardware levels, since blocks and grids are a specific solution for problem space mapping (Section 5) designed especially for NVIDIA GPUs and are not necessarily optimal for any hardware levels. In our solution, which might be considered as a modification and extension of the OpenCL model, a memory transfer API is defined for each levels at the execution layer and split-merge-exchange operations are defined with address translator at the problem space layer. This API of the execution layer can be implemented in different ways including the one proposed in [Strøgert et al. 2008] or in the way we did (Section 6). The most important benefit is that the code at the highest abstraction layer, at the problem space layer only has to be altered if the underlying hardware architecture significantly changes. For instance, when porting the application from GPUs to Cell BE.

In next three sections, we summarize the three different abstraction layers of our model. At the end of each section, a usage of the components of the model is illustrated by an example that is depicted in Figure 1.

3 Layer 1: A Conceptual Hardware Architecture Model

First, we introduce a simple formalism to describe hardware architectures. Our model follows the key ideas behind the OpenCL Platform Model [Kronos OpenCL Working Group 2008] but with the following assumptions:

- Any hardware element is reliable, the execution times of any task and memory transfers are predictable, and there is no fault tolerance introduced in the model.
- The hardware architecture is homogeneous, the hardware elements are identical on the same level and the transfer times are approximately also equal.

The abstract hardware element of the execution is the processing element (PE), which is a virtual scalar processor that can execute sequential code.

PEs might be bundled into processing arrays (PA) that provide instruments to organize the cooperation of the PEs by barriers, shared memory, or communication channels.

A processing device (PD) is the physical unit of the computing hardware for a certain hardware level: a collection of PEs, PAs, and memory spaces.

A processing unit (PU) is a task execution interface for a processing device. PE, PA, or even PD can represent the task execution interface for a specific device.

Within their device, PEs may access data from multiple memory spaces during their execution. A PE may have a private memory space that is inaccessible for other PEs. Each PA may have a local memory space shared among all PEs of a PA. Finally, PD can have a global memory space to which all PEs have access. Private memory and local memory may be implemented as registers or as dedicated regions of memory on the PD. Alternatively, the these regions may be mapped onto sections of the global memory.

The model of a multiprocessor of a NVIDIA GPU, a graphics board, a desktop computer with a quad-core CPU and four graphics
Figure 1: An overview of an imaginary distributed computing system built up from the basic components of our model. From left to right, the abstraction layers are depicted bounded by upright dark grey rectangles: the hardware abstraction layer, the abstract execution model, and the problem space layer, respectively. From bottom to top, the scalability of the system is enhanced with levels bounded by horizontal light grey polygons.

(a) Abstract model of multiprocessor from a NVIDIA GPU, a graphics board, an SMP computer, and a compute cluster. (b) Contexts and task instances of a kernel, a process, and a job corresponding to the hardware level. (c) A typical problem space partitioning scenario that illustrates split memory objects in system memory, in global memory of the graphics board and in local memory of a NVIDIA multiprocessor driven by address translation.

4 Layer 2: Execution Model

This section explains a logical execution model that can be mapped to the mentioned abstract hardware. In our work, we restricted ourselves to the class of problems with the following facts assumed:

- The execution control, including executing tasks and performing memory transfers only depend on the task to instantiate and the extent of the input and output data, and does not depend on the data itself. The size of the related data is always known in advance. This also implies that tasks are associated with data before the execution of tasks begins.

- Only task instances of the same task code can cooperate. This assumes at least SPMD or the stricter SIMD or SISD programming model at any level of the system.
The distributed parallel architecture operates on the same instance of a problem at the same time, not on multitudinous parallel transactions.

4.1 Logical Units of Execution and Storage

In this paper, we define a task as a sequence of instructions that operate together as a group corresponding to some logical part of a program. A task, that is implemented in a procedural programming language, has well defined inputs and outputs and it has to be executed on a PU on purpose to do some modification on its output data.

When a task is submitted for execution, a discrete index space is also defined by the caller. For each point of this index space, an instance of the task is executed, which is called work-item (WI). WIs are only distinguished from each other by their own index space positions.

Bundling WIs in a sub-space of the index space, work-groups (WG) force a coarser grained decomposition of the index space.

WIs are executed by single PEs maybe as part of a WG running on a PA. Each WI operates according to the same task but the actual sequence of instructions can be different across the set of WIs. WIs within a WG are executed concurrently and may interact through local memory objects and WG barriers implemented by the PA. The number of WGs can be usually higher than the number of PAs for a specific device as well as the number of WIs is usually not limited by the number of PEs in a PA.

Memory objects (MO) store linear collection of bytes in a memory space. Variables, arrays, textures, or data structures can also be considered as MOs. MOs are randomly addressable using pointers or maybe real valued positions by the target WIs. In contrast, MOs are usually not directly accessible by the source WI, which can manipulate them only by API calls.

Table 1 describes the properties of private MOs, local MOs, and global MOs defined in different memory spaces of the PD. Task instances of a job running on a compute cluster, instances of a process executed on an SMP thread, and instances of a kernel that are executed on a GPU is illustrated in Figure 1 (b).

<table>
<thead>
<tr>
<th>MO Type</th>
<th>Access</th>
<th>Scope</th>
<th>Lifetime</th>
<th>Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>private MO</td>
<td>sequential</td>
<td>one WI</td>
<td>execution of the WI</td>
<td>always consistent</td>
</tr>
<tr>
<td>local MO</td>
<td>parallel / sequential*</td>
<td>WIs in the same WG</td>
<td>execution of the WG</td>
<td>across WIs in the same WG only at a WG barrier</td>
</tr>
<tr>
<td>global MO</td>
<td>parallel / sequential*</td>
<td>any WI</td>
<td>managed by the source WI</td>
<td>across WIs in the same WG only at a WG barrier, for source WI only at command queue barrier</td>
</tr>
</tbody>
</table>

There are three types of commands:
1. An instantiation command creates WI task instances and WGs by defining index space for a task.
2. Memory space manipulation commands can be used for creating and destroying memory objects on the target PU and for interaction between the memory objects of source WI and the created target memory objects.
3. Synchronization commands enable to constrain the order of command executions. Command-queue barriers ensure that all previously queued commands have finished their execution and any modifications on global memory objects are effectively updated.

A GPU context maintained by a process WI, a node context managed by a job instance, and a cluster-context attachment point is illustrated in Figure 1 (b). An application pseudo-WI can be attached to any of these contexts: to the GPU context in case of single-GPU programs, to the node context for multi-GPU environments, and to the cluster context to get a GPU cluster system architecture.

5 Layer 3: Problem Space Mapping and Virtualization

Using the elements of the execution model, parallel and distributed tasks can be designed that is executed on an abstract multi-level hardware architecture. Nevertheless, reimplmenting a parallel algorithm originally designed for the shared memory architecture of a single GPU, on a distributed memory architecture is usually not a trivial “porting issue”. In general, it can be performed easily only for the simplest classes of problems – when the parallel task instances (e.g. kernel executions for the GPU) work on different data without any communication. Even so, suitable guidelines can be defined for the more general problem domain defined in Section 4 that enables detaching the memory and address space management from the algorithmic code. This section explains, how it can be done.
introduce this abstraction here as a third layer instead of trying to force into the second one.

Using this abstraction, the memory and address space management and the algorithmic code can be separated. This enables the virtualization of hardware resources of multiple PUs for the source WI, and it also makes possible the virtualization of the execution environment for the target WIs that can be exploited for writing reusable task codes. This is advantageous since tasks usually have very similar implementation that can be usually fused by defining a simple memory address translation for the input/output data.

In the terms of our model, an abstract task solves a certain computational problem on well defined data. For instance, an abstract task can be multiplication of two sparse matrices, sorting a sequence, or reconstruction of a volume from its X-ray projections. An abstract task refers to a set of different tasks that implement the functionality of the abstract task, and a control logic that selects and parameterizes one of these tasks for a given abstract hardware configuration and the specified extent of the input/output data.

Dealing with computational problems with high memory requirements, the Geometric Decomposition design pattern [Mattson et al. 2005] is an appropriate solution to the recurring problem of execution organization and managing data decomposition among multiple WIs of the adjacent levels of parallelism. In this case, the problem space is partitioned into chunks, where the solution for each chunk can be solved using data from only a few other chunks. This scheme requires the following operations to be implemented for a certain task:

- The global problem space is partitioned into coarse-grained chunks by the split and merge operation pair considering the shape of the chunks. Decomposition can be performed using write and read memory copies or memory mappings.
- The exchange operation ensures that each task has access to all data needed for its subsequent execution by target-to-target data transfer.
- The computation is done by the execute operation, that updates the data within the chunk by instantiate WIs on the target PU.

Using these pattern elements the generic outline of a task is

\[
\text{split} \rightarrow \text{execute} \rightarrow \text{[exchange \rightarrow execute]* \rightarrow merge} \quad (1)
\]

where the asterisk indicates that \([\text{exchange \rightarrow execute}]\) block might be executed zero or more times. This model can be recursively applied since the \text{execute} operation invokes another task on the target PU.

To efficiently encapsulate the data decomposition strategy for both memory transfer operations like split, exchange, merge, and for unifying the environment for the called WIs we apply \text{address translators} (AT) [Lefohn et al. 2006]. An AT is an object for memory read and write operations supporting both \text{random-access} reads and writes, and \text{range translations} to support efficient memory copy operations.

Technically, only the index space splitting scheme have to be specified by the programmer, since it is closely related to the solved problem. We think there is no framework that can do it automatically, thus in our solution just a structured way is defined for the programmer to express it. Nevertheless, the number of practical splitting schemes is usually low, usually cloning or modulo partitioning (for instance into stripes or chessboard cells) is appropriate. This index space splitting scheme then directly implies the automatic execution of instantiation and memory space manipulation commands.

As an example, distribution of the problem space among job instances, SMP processes, and GPU kernel instances is depicted in Figure 1 (c). Listing 1 illustrates code snippets of an imaginary abstract task and one of its implementation. The specific task implementation of the abstract task is selected based on the hardware resources and extent of the input/output data. The hardware resources and the input/output data also rules the selection and the parametrization of the split/merge parameters and the address translators.
6 Implementation Details

In order to illustrate a realization of our model, we built a system of connected desktop computer with multi-core CPUs and modern GPUs according to the abstract hardware architecture outlined in Figure 1 (a).

6.1 Hardware Layer

At the topmost level of scalability, the cluster PEs, i.e. the nodes of the cluster can cooperate via their network adapters. The cluster PU is represented by the cluster itself, the cluster PD.

On the next level, processor cores can be considered as SMP PEs since their access to the system memory is symmetric. The SMP PDs have memory spaces, in our case 4 GB system memory fully addressable by any SMP PE. Since the L1 caches are not addressable directly, no SMP private memory exists, while the SMP local and SMP global memory spaces coincide. The SMP PA stands for SMP PU, the multi-core CPU when using homogeneous processor threads using OpenMP for instance. Note that SMP PE can at the same time be SMP PU, e.g. when using pthreads.

On the lowest level of scalability, stream PAs are located in multiprocessors on the graphics board. There are two additional cache spaces over the 16 KB local store in each multiprocessor: the constant memory cache and the texture cache. These caches can be fully exploited when using CUDA interface for programming the graphics device. A stream PD which the the stream PU consists of numerous stream PA, 16 NVIDIA multiprocessors for our graphics cards, while each multiprocessor contains 8 scalar processors, stream PEs. There is an additional stream global memory space for each stream PD, 512 MB and 1 GB in our case. See Figure 1 (a) and Table 2.

6.2 Execution Layer

According to the scalability levels, we defined three task levels for the execution layer as it is illustrated in Figure 1 (b) and Table 2. We have implemented our code in C++ under Linux operating system.

To invoke the cluster WIs, the job instances at the topmost level, we applied Grid Engine scheduler. The communication between the job instances is performed using MPI2 calls, for which we used the OpenMPI implementation. For the simplicity, no cluster WGs are defined, nevertheless using MPI2 it is also possible if needed.

On the next level of scalability, OpenMP is applied to spawn SMP WIs using pragma compiler directives in the C++ code. The OpenMP threads are grouped into thread teams as SMP WGs. As a straightforward solution, shared system memory is applied for communication between SMP WIs. However, significantly different implementations are also possible, for instance pthreads with message queues, if it fits better to the demands.

Finally, the runtime API of CUDA implements the lowest scalability level. According to the recommendation of NVIDIA, a dedicated SMP WI is assigned to each stream PU. This SMP WI manages the memory operations and the kernel invocations. For now, according to our assumptions about the homogeneous execution (Section 4), each stream PD executes the same kernel at the same time. The stream WIs correspond to CUDA kernel instances with specific group and block indices, while stream WG represents a CUDA block in which synchronization and fast communication is possible.

7 An Example and Evaluation

A problem space layer has to be defined in order to evaluate the implementation of the hardware and execution layers. For this purpose we implemented the GPU adaptation of a complete practical helical CT reconstruction scheme with projection image preprocessing, filtering, and back-projection. Our implementation is based on the well-known FDK method [Feldkamp et al. 1984], which solves the three-dimensional reconstruction for cone-beam CT scanners. To avoid the CPU to become the bottleneck of the system, not only the filtering and the back-projection step of the FDK method, but also each step preprocessing the scanned raw projections are decomposed into parallel tasks performed on the GPU as different kernel programs such as offset, gain, bad pixel, and geometrical corrections.

In the highest level of abstraction, projection images are cloned for each PU in the split phase. Therefore, no address translation is needed for the projections. 2D image processing tasks are executed on each GPU since according to our experiences, the execution time of these tasks is not dominant in the overall execution time (Table 3). To avoid the write-write collisions the index space is directly mapped to the output volume for the back-projection. The output volume is split among the target GPUs and the reconstructed data have to be copied back by the source WI. Due to the helical geometry of the scanner not every projection has to be projected back to any voxel. In order to balance the computational workload among the GPUs, axial slices are assigned to GPUs in a modulo scheme. To support this scheme, a simple memory translator was implemented that creates a mapping between the logical voxel coordinates of the problem space and the memory address to the stored sub-volume data.

For our experiences we used a Mediso NanoScan™ CT scanner and a LEGO® Technic plastic figure as a specimen (see Figure 2 for the reconstructed image). We have investigated two different hardware configurations:

1. Two desktop computers, each equipped with an Intel Core2 6700
The volume was reconstructed on a Cartesian Cubic lattice in four resolution settings (384 MB, 768 MB, 1.5 GB, and 3 GB) with single precision floating point values from 360 projection images each consisting of 1024 × 2048 pixels with 16-bit values (1.4 GB). Nine rays were cast for each voxel.

The projection transfer times, the 2D correction and filtering times, the back-projection time, and the volume transfer times as well as the performance scaling values are detailed in Table 3. The task execution times were monitored using GNU gprof and using NVIDIA CUDA Profiler. These values reflect the efficiency of our C++ and CUDA implementation of the tomographic reconstruction algorithm. The computational cost of the implementation of our model elements is practically zero since these task have to be performed at any cases: creating light-weight CPU threads, copying memory to the device, launching CUDA threads. The “overhead” is 2-3 stable fetches for split, merge, exchange, and execute calls since C++ virtual methods are introduced to detach the implementation from the API.

The network transfer times over Gigabit Ethernet are constant 15 – 16 seconds for the projection images of 1.4 GB which is higher than the sum of all other steps for config-2, 2-PC, 2-GPU case for tiny and small volumes and still dominates the overall time for big cases, too. Using Infiniband network architecture these transfer times can be lowered by a magnitude and can be comparable to the PCI Express transfer times. Thus, these circumstances have to be considered when evaluating the overall speedup factors of this implementation on this hardware.

On the other hand, since all addressing calculations are encapsulated in the address translator logic, a volume of three gigabytes can be reconstructed in a minute using config-2 without any modification of the C++ or CUDA code, only with tuning the parameters of the execution.

The rendering of the reconstructed “small” data set is depicted in Figure 2. The image was generated using a simple OpenGL program rendering a full-screen quad with a Cg pixel shader performing first hit ray casting in a 3D texture.

Figure 2: Rendering of the reconstructed volumetric data of a LEGO® Technic plastic figure with a pipe wrench in his hand using first hit ray casting with a simple local illumination model.

dual-core CPU @ 2.6 GHz, 4 GB of DDR2 RAM, and two NVIDIA 9800X2 GPU (2 × 512 MB RAM).

2. Two desktop computers, each equipped with an Intel Core2 Q9450 quad-core CPU @ 2.6 GHz, 4 GB of DDR3 RAM, and two NVIDIA GTX-280 GPU (1 × 1024 MB RAM).

In both cases, the desktop computers were connected over Gigabit Ethernet using Intel Pro1000 network adapters. Actually, Gigabit Ethernet is not suitable as the interconnection network of a compute cluster. We could hardly reach 90-100 MB/s bandwidth with the average delay of 1-3 milliseconds and 1-1.5% packet lost ratio. Thus, in order to increase effective network transfer rate we used MPI2 only to control the distributed application. For transferring data we have created a custom solution using simplex point-to-point connection with 8k jumbo frames and UDP protocol. Nevertheless, it is worth to use InfiniBand or Myrinet technology for this purpose. Running tests on such systems remains our future work.

The volume was reconstructed on a Cartesian Cubic lattice in four resolution settings (384 MB, 768 MB, 1.5 GB, and 3 GB) with single precision floating point values from 360 projection images each consisting of 1024 × 2048 pixels with 16-bit values (1.4 GB). Nine rays were cast for each voxel.

The projection transfer times, the 2D correction and filtering times, the back-projection time, and the volume transfer times as well as the performance scaling values are detailed in Table 3. The task execution times were monitored using GNU gprof and using NVIDIA CUDA Profiler. These values reflect the efficiency of our C++ and CUDA implementation of the tomographic reconstruction algorithm. The computational cost of the implementation of our model elements is practically zero since these task have to be performed at any cases: creating light-weight CPU threads, copying memory to the device, launching CUDA threads. The “overhead” is 2-3 stable fetches for split, merge, exchange, and execute calls since C++ virtual methods are introduced to detach the implementation from the API.

The network transfer times over Gigabit Ethernet are constant 15 – 16 seconds for the projection images of 1.4 GB which is higher than the sum of all other steps for config-2, 2-PC, 2-GPU case for tiny and small volumes and still dominates the overall time for big cases, too. Using Infiniband network architecture these transfer times can be lowered by a magnitude and can be comparable to the PCI Express transfer times. Thus, these circumstances have to be considered when evaluating the overall speedup factors of this implementation on this hardware.

On the other hand, since all addressing calculations are encapsulated in the address translator logic, a volume of three gigabytes can be reconstructed in a minute using config-2 without any modification of the C++ or CUDA code, only with tuning the parameters of the execution.

The rendering of the reconstructed “small” data set is depicted in Figure 2. The image was generated using a simple OpenGL program rendering a full-screen quad with a Cg pixel shader performing first hit ray casting in a 3D texture.

### Table 3: Mean values of task execution times in seconds for different volume sizes on different hardware architectures. Task executions include computations and memory transfers, through the network and via the PCI Express bus.

<table>
<thead>
<tr>
<th>volume size</th>
<th>hardware architecture</th>
<th>projection transfer</th>
<th>2D image processing</th>
<th>back projection</th>
<th>volume transfer</th>
<th>total</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>tiny, 384 MB 512 × 512 × 384</td>
<td>1 PC</td>
<td>1 GPU</td>
<td>0.7</td>
<td>5.9</td>
<td>115.0</td>
<td>0.3</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2 PCs</td>
<td>2 GPUs</td>
<td>-</td>
<td>2.8</td>
<td>5.9</td>
<td>26.0</td>
<td>0.4</td>
</tr>
</tbody>
</table>

| tiny, 384 MB 512 × 512 × 384 | 1 PC | 1 GPU | - | 0.6 | 3.7 | 25.0 | 0.3 | - | 29.6 | × 1.00 |
| | 2 PCs | 2 GPUs | - | 1.2 | 3.7 | 12.6 | 0.2 | - | 18.2 | × 1.73 |

| small, 768 MB 512 × 512 × 768 | 1 PC | 1 GPU | - | 0.6 | 3.7 | 46.0 | 0.6 | - | 50.9 | × 1.00 |
| | 2 PCs | 2 GPUs | - | 1.2 | 3.7 | 23.0 | 0.6 | - | 28.5 | × 1.86 |

| medium, 1.5 GB 1k × 1k × 384 | 1 PC | 1 GPU | - | 0.6 | 3.7 | 24.9 | 0.3 | - | 47.3 | × 1.73 |
| | 2 PCs | 2 GPUs | - | 1.2 | 3.7 | 11.0 | 0.2 | - | 33.8 | × 3.16 |

| large, 3 GB 1k × 1k × 768 | 2 PCs | 2 GPUs | - | 1.2 | 3.7 | 41.4 | 1.2 | - | 47.5 | × 1.00 |

The rendering of the reconstructed “small” data set is depicted in Figure 2. The image was generated using a simple OpenGL program rendering a full-screen quad with a Cg pixel shader performing first hit ray casting in a 3D texture.
8 Conclusion and Future Work

We have presented a scalable, multi-layer and multi-level programming model for recent GPU-based high performance computing systems. This model different aspects of a computing system can be described. It enables creating an abstract architectural model for different hardware architectures as well as defining tasks from basic components of the execution model that can be mapped to the hardware model using contexts implementations. Instead of hiding the execution environment from the programmer, we define a higher abstraction level on the top of the execution model to provide abstract problem space mapping with code reusability and virtualization of hardware resources.

We have created a C++ implementation of the abstract model elements and we also defined contexts, work-groups, and work-item implementations using recent popular parallel programming frameworks: OpenMPI, OpenMP, and CUDA. Within this programming environment we have implemented a medical industry level micro-CT reconstruction application and evaluated its performance scalability. In the future, we plan to evaluate the flexibility and the performance of our model on more applications especially when multiple split-exchange-merge strategies can be investigated and compared.

We plan to support OpenCL contexts as well as this technology seems to be an upcoming standard. On the other hand, there are several ways to ease the constraints we made during the design of the model. Some of these are software considerations, such as introducing features of transaction systems enabling the distributed parallel architecture to execute multiple instances of the same application or instances of different applications. For SMP threads we used homogeneous task instances, although it is not necessary. On the other hand, the execution control could be be data-dependent as the size of the input/output data might be unknown a priori. From the hardware point of view, supporting inhomogeneous hardware architectures as well as fault tolerant systems can be a future work.

Acknowledgements

This work was supported by the Hungarian National Office for Research and Technology (TECH08A2) and Mediso Medical Imaging Systems.

References


